Investigation and Implementation of Available Software and Algorithms for Autonomous Vehicle Development

Honors Undergraduate Research Thesis

David Flessner

The Ohio State University

Department of Mechanical and Aerospace Engineering

2020

Thesis Committee

Professor Levent Güvenç, Advisor

Professor Bilin Aksun- Güvenç

Abstract

The purpose of this project is to integrate available algorithms for automated driving from open source software into a simulation to demonstrate how the software can be used to quickly prototype control systems for automated vehicles. Autoware is the primary software program that will be tested which processes incoming sensing data for automated driving functions resulting in control inputs for the vehicle such as speed and steering control to respond to the various driving situations. Most of these programs for automated driving research are built based on Robot Operating System (ROS) which is an open source meta-operating system specifically designed to be used for robot software development. Testing of Autoware was completed with ROS visualization simulation tools in addition to a software program called LGSVL Simulator based on the Unity game engine which was used to develop a variety of testing scenarios. Once the software integration was completed, performance of the different algorithms was evaluated using LGSVL simulator and scenarios built in Unity. The primary goal of the project was to uncover the functionality of open source autonomous vehicle software platforms and develop realistic simulation scenarios for the testing of the algorithms.

Acknowledgements

I would like to thank my advisor Professor Levent Güvenç for allowing me to take part in this research and connecting me with the people and resources of the laboratory to enrich my undergraduate experience. I would also like to thank Professor Bilin Aksun-Güvenç for serving on my defense committee as well as the graduate students at the lab particularly Sukru Yaren Gelbal and Xinchin Li for taking time to guide me through this.

Table of Contents

List of Figures

Chapter 1: Introduction

1.1 State of Autonomous Driving

SAE International defines autonomy on a range of levels 0-5 with level 0 being completely non-automated to level 5 being completely autonomous. Commercial vehicles currently operate under level 2 autonomy which is considered partial automation while the most advanced research vehicles including autonomous shuttles fall under level 4 automation where the vehicle can operate autonomously under certain conditions without driver intervention. Significant advances in recent years resulted in Audi originally aiming to sell the first SAE Level 3 autonomy production vehicle with Traffic Jam Pilot, but legal and infrastructure issues prevented Audi from releasing the technology with the vehicle [1]. Many manufacturers are bypassing level 3 and claim level 4 and fully autonomous driving production vehicles will be available by the early 2020's [2]. Additional efforts are also being made to enable cooperative autonomous driving through the use of V2X communication with platforms such as the CARMA developed by the Federal Highway Administration [3]. This communication would enable autonomous vehicles to utilize much more of their potential by creating many more nodes of information for the vehicles to take advantage of to uncover a much clearer picture of the situation that the vehicle is encountering. The possibility for cooperative autonomous driving maneuvers such as platooning could also be implemented for the vehicles to operate more efficiently and mitigate traffic congestion by damping traffic shock waves.

A closer look at the research going into automated driving will reveal that many open source software programs have been created and are still being advanced to aid in the development of automated driving software. Some programs already contain algorithms that can be used to

control speed and steering of vehicles which allows rapid development of automated systems for various vehicles.

1.2 Autoware and ROS

Autoware is the open source software program analyzed in this research which has been extensively used in research and industry with automotive manufacturers considering it as a baseline for prototyping automated vehicles [4]. The original Autoware program is used by more than 100 companies and is supported by the largest autonomous driving open source community [5]. ROS as the foundation allows Autoware to utilize ROS's modularity to develop and test software components separately. This enables the open source community to develop software packages in parallel so that rapid prototyping can be accomplished. The structure of how the software interfaces is shown in Figure 1.

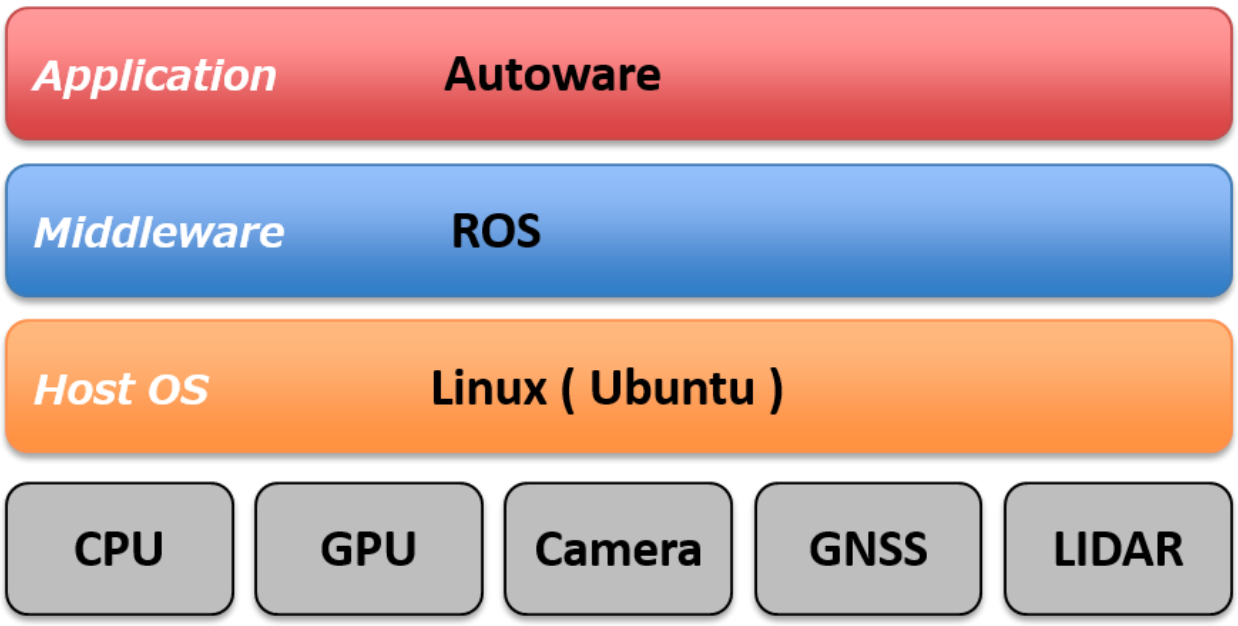

Figure 1: General Framework [6]

The modules that Autoware contains to enable automated driving include sensing, localization, detection, prediction, planning and control. The main sensors that Autoware supports are LiDAR, GPS, IMU and camera, and for each, a large variety is supported. For localization, a few algorithms are available, but primarily the Normal Distributions Transform (NDT) algorithm with LiDAR data is used to obtain the vehicle's position relative to a 3D point cloud map. The LiDAR data can additionally be combined with data from other sensors to more accurately localize the vehicle. For object detection, Autoware contains a variety of algorithms using sensor fusion of LiDAR and camera data to accurately detect and track objects while simultaneously minimizing the computation time. This data can also be combined with Kalman and particle filters to predict the trajectories of moving objects. Once these detection and prediction modules are operating, Autoware's decision module uses an intelligent state machine to manage mission planning and motion planning which generate global and local trajectories respectively. With the trajectories generated, actuation is completed with the pure pursuit algorithm to create discrete waypoints for the path following and velocity and angle commands for a controller to then actuate the steering and throttle. This brief overview of the core modules encompasses the general functionality of Autoware, but the primary focus for this project was Autoware's decision maker package.

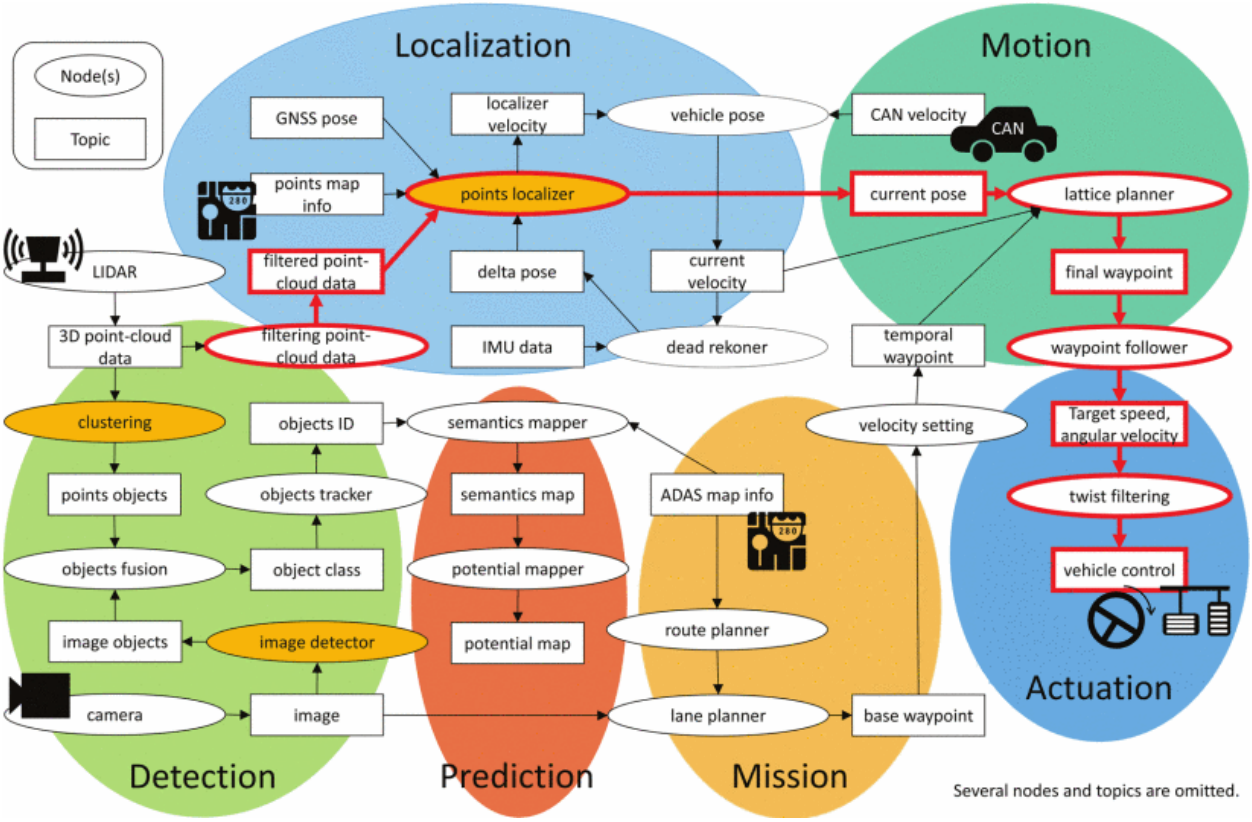

Figure 2: Node and Topic Configuration of Autoware [3]

1.3 Decision Maker

The decision maker of Autoware coordinates with the state machine to manage the Vehicle status using information from the Mission Status, Driving Status, and Behavior Status. These statuses decide how the vehicle should operate and which software components to use through a finite state machine defined for each status. Decision making is a vital component in any autonomous system; however, much of Autoware's decision maker is abstracted into code, so more documentation on the inner workings would be beneficial to developers. Accordingly, one of the objectives of this project is to provide supporting documentation to aid in Autoware development.

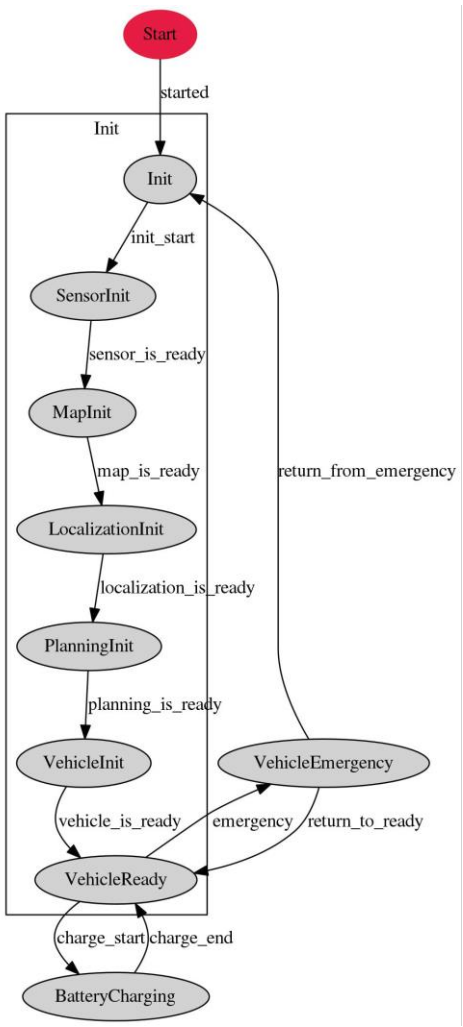

Figure 3: Vehicle States [7]

Displayed above is the state diagram for the vehicle status. This status is used for initialization and monitoring to ensure the vehicle's systems are functioning properly. This status cycles through the main modulus required for automation to ensure all the required information is being communicated through the different ROS topics. In Figure 4 below, is the state flow diagram for the Mission Status. This status is used for mission planning of global trajectories and navigation once the mission order is received and compatible.

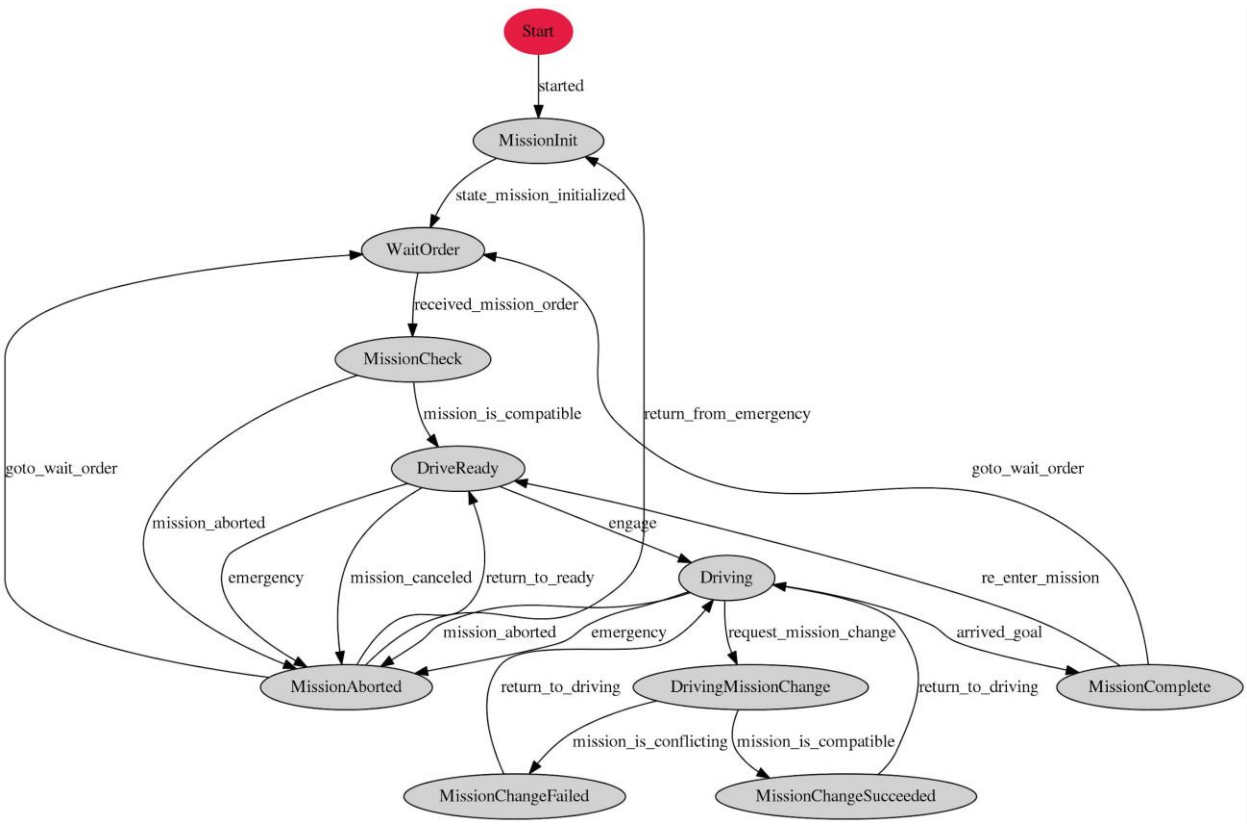

Figure 4: Mission States [7]

Finally, below is the motion state flow diagram. Until the vehicle motion state is DriveReady, this status is waiting, but when the vehicle is drive ready and engaged, this module manages all the necessary local trajectory planning and control while driving.

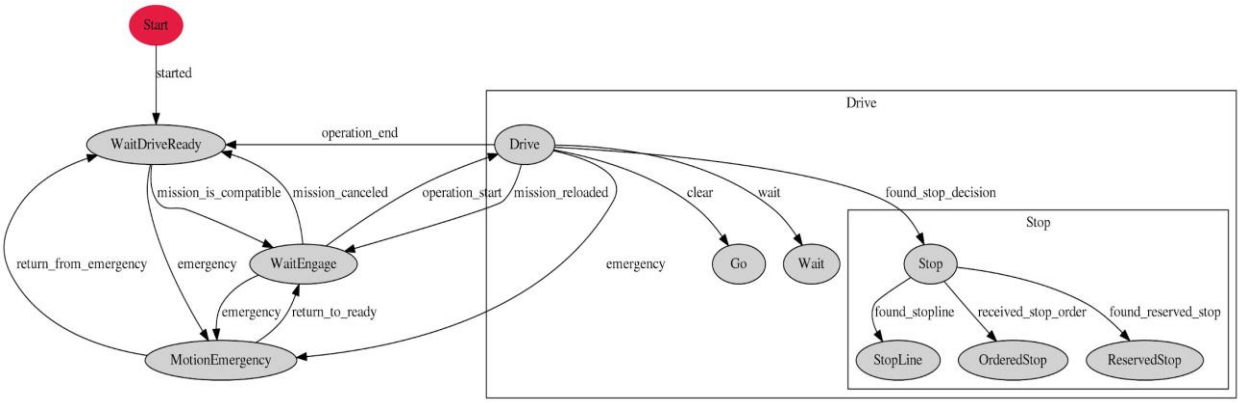

Figure 5: Motion States [7]

The test bed used was the LGSVL Simulator developed by the LG Electronics America R&D Center using the Unity game engine. This simulator enables full integration with Autoware and being based in Unity, allows extreme customizability of environmental and vehicle parameters. In the simulator, an autonomous shuttle route in Columbus, Ohio known as the Linden route was recreated and developed as a part of the Automated Driving Lab's Smart Campus project of building and using realistic simulation environments for developing and testing autonomous driving algorithms. This environment built for the Linden route was used as the base of the scenes built with Unity [8] [9].

The broad outcome from this project is to be able to understand the development of software for automated vehicles so that more complex concepts and algorithms can be developed and applied in a similar fashion to increase the performance of the automated vehicles. More specifically, a deeper understanding of the main software that is used with automated vehicle development and simulation will be gained.

Chapter 2: Methodology

2.1 Overall Approach

Investigation into Autoware's decision maker was completed using existing documentation for Autoware, ROS tools included in Autoware, and assessing the code. Demo data available online was used to run simulations in addition to scenes developed in Unity. Next, integration with the LGSVL Simulator was completed with Autoware connection over ROS bridge. Finally, The Unity development served a twofold purpose of testing Autoware as well as exploring Unity as a software to accurately create simulation environments. These scenarios were built in Unity then integrated with Autoware to test Autoware's functionality.

2.2 Operating Decision Maker

To begin to work with Autoware's decision maker or Autoware at all, a source install of Autoware was completed from Autoware's GitHub page using the available documentation. Sample data available on the page was used to read sensor data into Autoware to evaluate Autoware's response from the computing packages. The documentation available for the decision maker was then followed to get the package functioning properly with the proper ROS topics publishing that it needed to subscribe to. The RQT graph tool available from Autoware's runtime manager was then used to view the different topics the package was subscribed and publishing to at different states of the main statuses.

2.3 Autoware Integration with LGSVL Simulator

After the decision maker analysis was completed, integration of Autoware with LGSVL Simulator was completed by following documentation available on the LGSVL Autoware GitHub repository. After proper setup and software troubleshooting, the ROS bridge was able to

connect the two and operate the simulator with functionality from Autoware. From this, path following tests and analysis into Autoware's pure pursuit follower was conducted.

2.4 Unity Development

Starting with a prebuilt scene of the Linden route, Unity was used to create collision scenarios of vehicles in straight crossing paths at non-signalized junctions, a vehicle following scenario with the lead vehicle suddenly stopping, and a pedestrian crossing scenario. Each of these situations were taken from NHTSA's list of pre-crash scenarios as possible scenarios that could be encountered on the Linden route [10]. Additionally, a scenario was built for the navigation of roundabouts as another test case for the autonomous vehicle.

Chapter 3: Results and Discussion

3.1 Decision Maker Analysis

The Decision Maker package was the primary focus of the investigation into Autoware. Using Autoware with ROS's RViz and RQT, ROS topic diagrams were able to be created at various states of Autoware's operation to understand how the software was functioning. On start, the Vehicle state transitioned through an initialization process allowing for sensors, maps, localization and planning to be initialized while the other statuses remained WaitOrder for Mission and WaitVehicleReady for Motion. This Vehicle initialization was essentially the decision maker waiting for all the required topics to begin publishing that it needed to subscribe to. All of these states had essential programs that allow the vehicle to read and position itself in its environment. Once all the initialization was completed, the Vehicle state changed to VehicleReady signifying that it is localized and ready for its Mission order. In Figure 6 below, the topics the decision maker was subscribed to and published to while the Mission state remained WaitOrder can be seen.

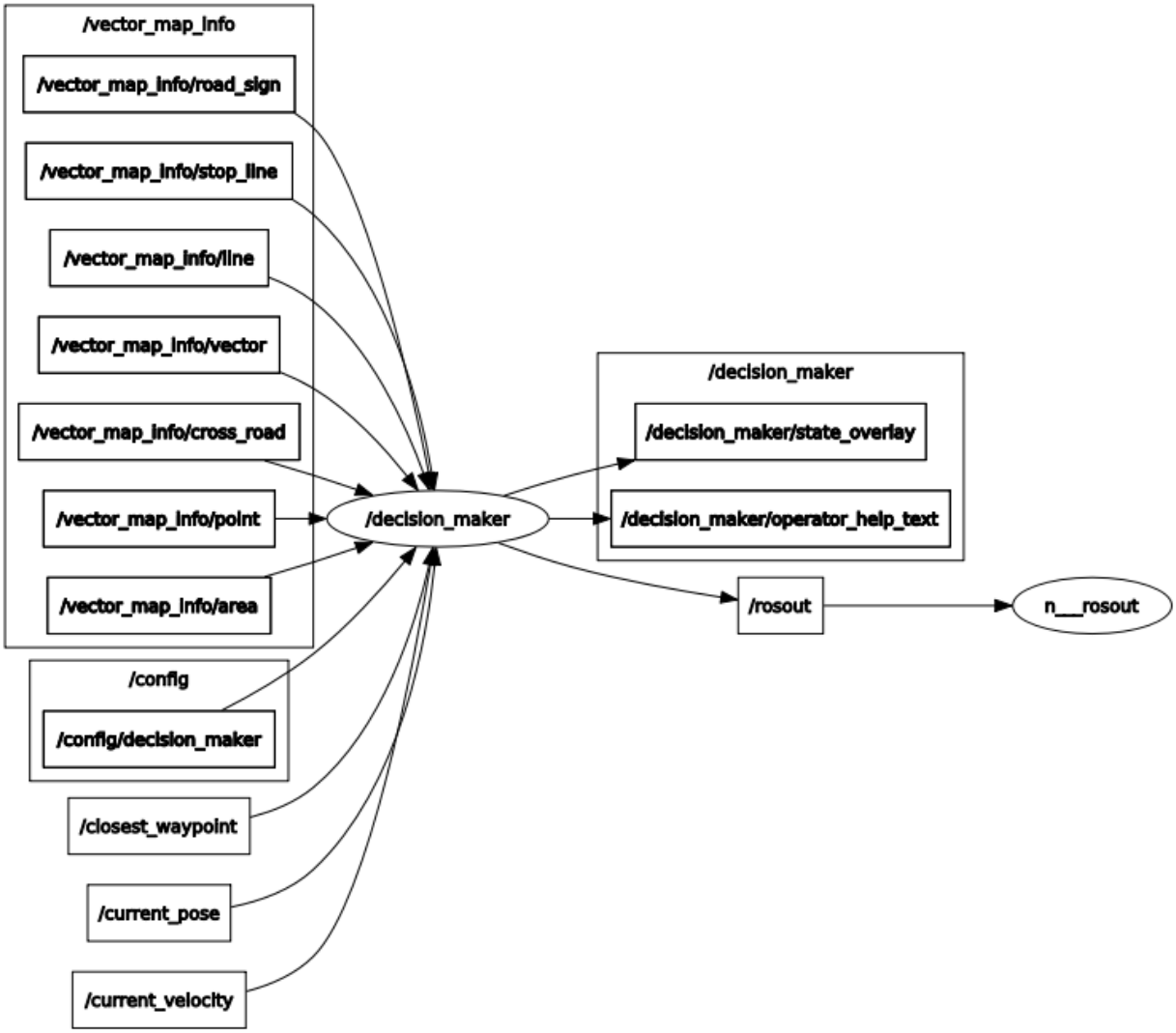

Figure 6: VehicleReady, WaitOrder, WaitDriveReady

Conceptually, the publishers were the map data needed for obeying traffic rules, velocity and position for localization, and the closest waypoint to begin planning. The only subscribers to the decision maker at this point were the RViz overlays which allowed textual visualizations for each status and a help monitor. Once a motion planner had been specified and the mission was compatible, the decision maker began publishing the following: a stop waypoint index to the state machine so it knows the end of the mission, the current state of each status to the decision maker

so it could update as needed, a traffic light status, an array for the lane waypoints to update the planner, and lamp commands for turn signals on the vehicle. The immediate subscribers and publishers to the decision maker in these states are shown in Figure 7, and additionally, a more complete diagram of the flow of information and the feedback is shown in Figure 8 with topics publishing to decision maker colored blue and topics subscribed to colored green.

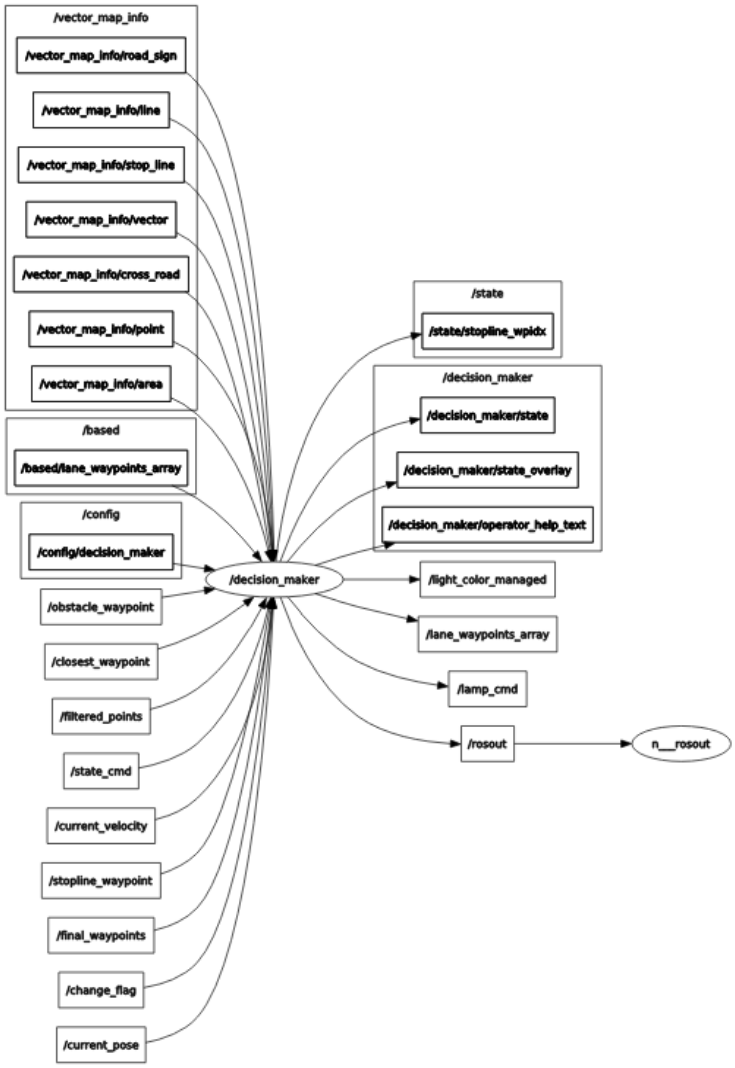

Figure 7: VehicleReady, DriveReady, WaitEngage

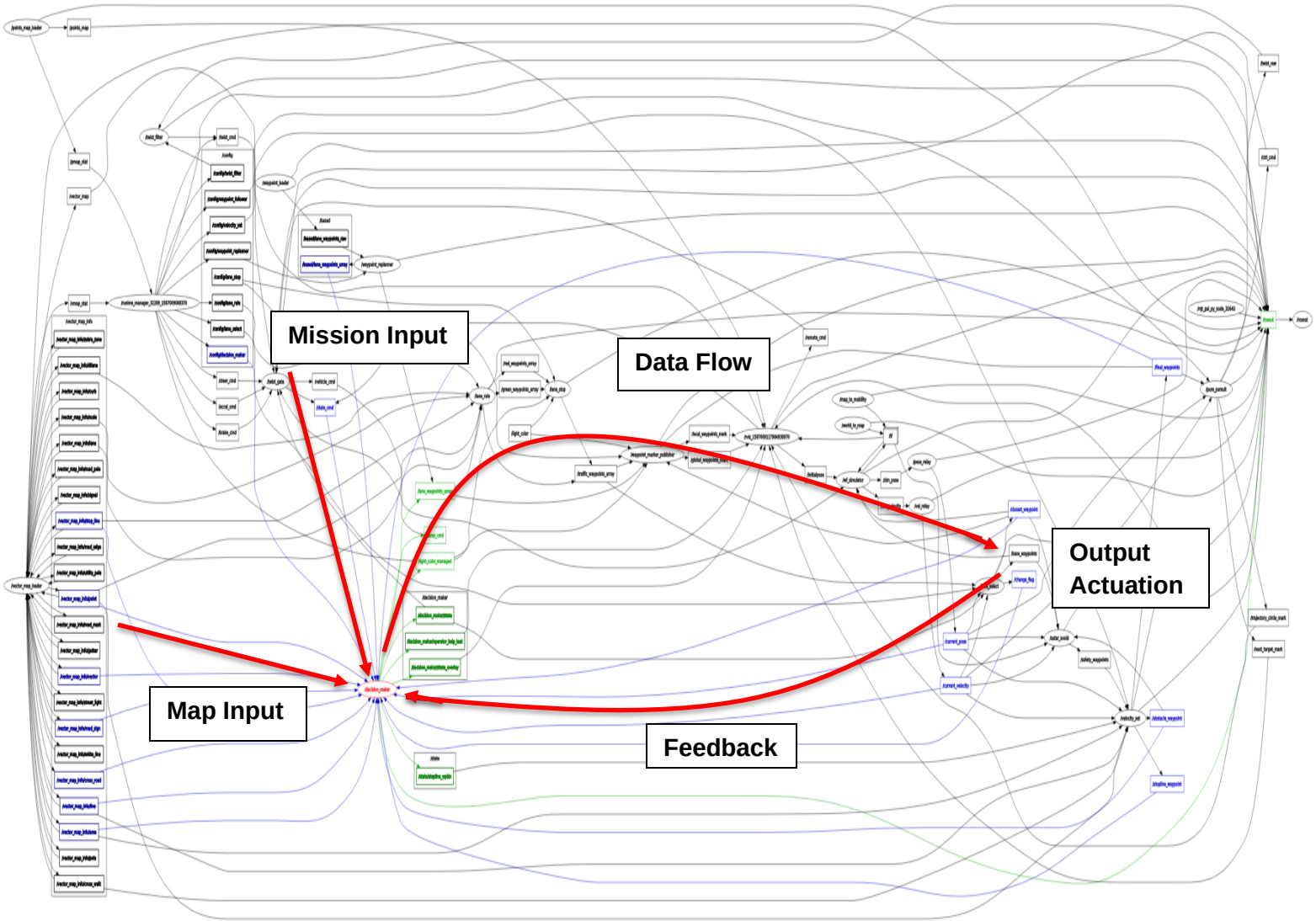

Figure 8: Complete Data Flow with VehicleReady, DriveReady, Driving

Looking at the complete data flow, the main feedback loop took place with map and mission data as the reference. These inputs manifested themselves as the vector map and the waypoint array for the vehicle to follow. The primary data used for planning that was passed on was the waypoint array which the following planners translate into actuation for velocity and position control which was then fed back to the decision maker. The nested design of the state machine allowed the feedback loop to function efficiently by limiting the flow of information while also updating conditions to transition between states as necessary. It also allowed for more efficient development with state definitions confined to one status so that the entire state machine would not have to be redesigned for an additional state.

3.2 LGSVL Integration

Autoware integration with the LGSVL simulator was completed after some software troubleshooting. Once both software programs were properly set up, connection over the ROS bridge was achieved for communication between the two. Displayed below is the RViz result of Autoware bridging to LGSVL's Borregas Ave map and operating with a path following scenario.

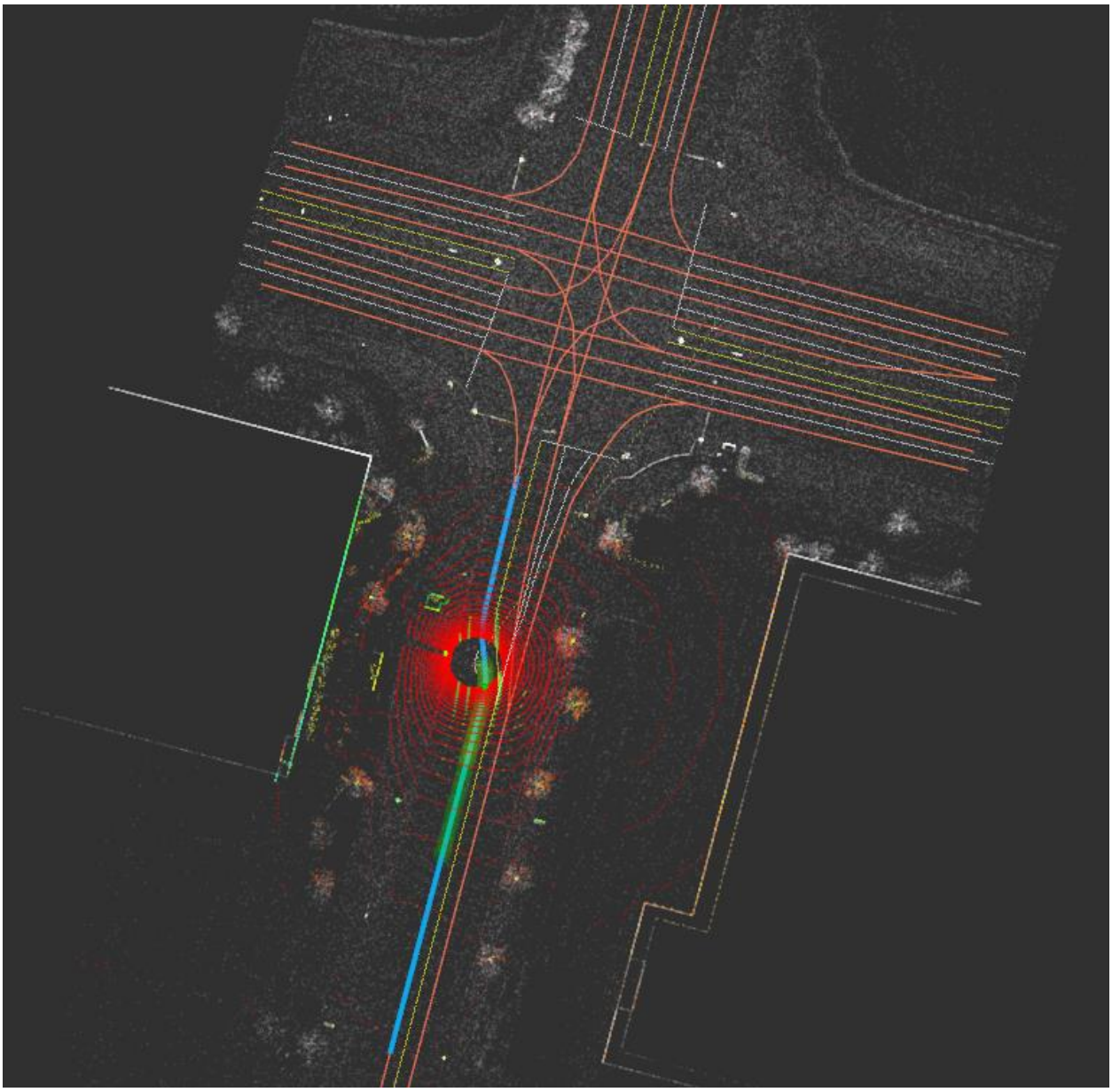

Figure 9: RViz Visualization of LGSVL Integration with Autoware

The RViz visualization allows the user to view vector map data including lane lines and the orange lane centerlines used by the mission planner. Once localization is completed, a navigation goal can be set and if the goal is valid with the appropriate planners, the vehicle will plan global and local trajectories in accordance with the traffic rules provided by the lane data. The global trajectory is visualized by the blue line over the center lane line, and the local trajectory

planning is represented by the green waypoints in front of the vehicle that discretize the path in order to give a reference signal to the planner. Additionally, the sensing data from the LiDAR was visualized with the concentric red lines. The localization module used this data to locate the vehicle with respect to the point cloud map.

Using the default demonstration algorithms to follow the path seen in Figure 9 above, the vehicle was able to accomplish the mission, but large deviations in the path were present in the slight bends resulting with the vehicle breaking the side lane line.

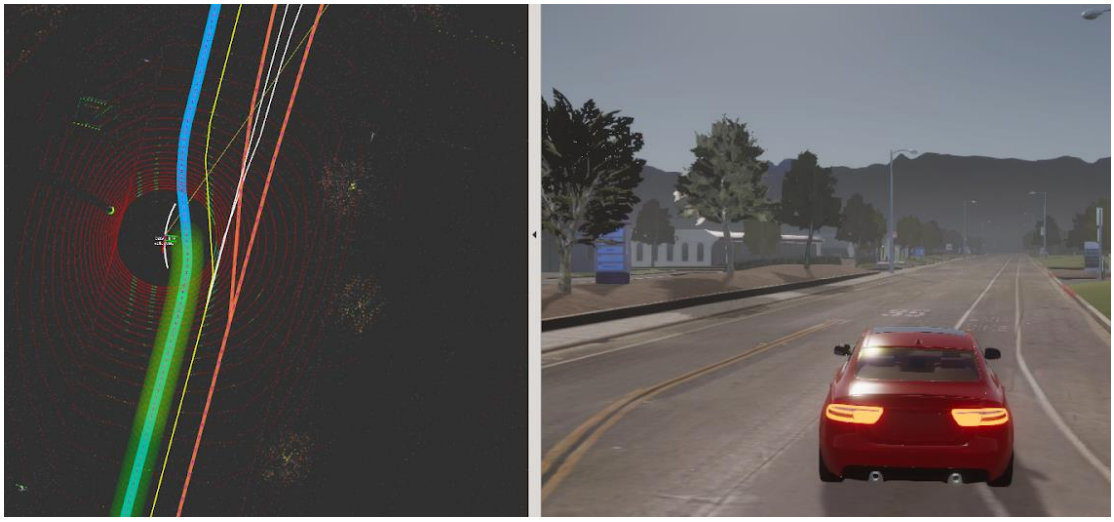

Figure 10: Initial Path Deviation

This result was from the pure pursuit follower not being properly tuned. This algorithm essentially sets a waypoint some distance in front of the vehicle to follow, so if the vehicle is traveling at low speeds with a waypoint too far in front of it, the vehicle will not follow the path properly [11]. The default settings for the follower used for steering control was observed to be not aggressive enough with the vehicle not actuating sufficient steering to follow the turn later resulting in oscillations after the initial deviation to such an extent that the vehicle broke the centerline of the road.

Figure 11: First Oscillation after Deviation

The lookahead distance parameter of the pure pursuit follower which plans the local trajectories was tuned from 4 to 2 waypoints to lookahead for the vehicle to follow the path better at low speeds while still avoiding constant changes that were present when the lookahead distance was set to 1. Seen in the next two figures is the improvement resulting from tuning the follower.

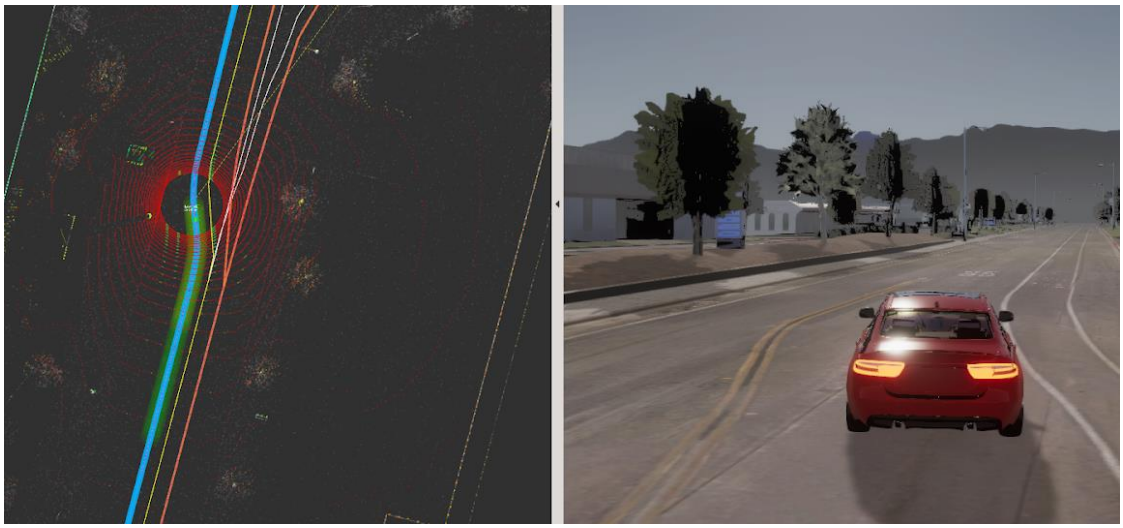

Figure 12: Initial Path Deviation with Tuned Follower

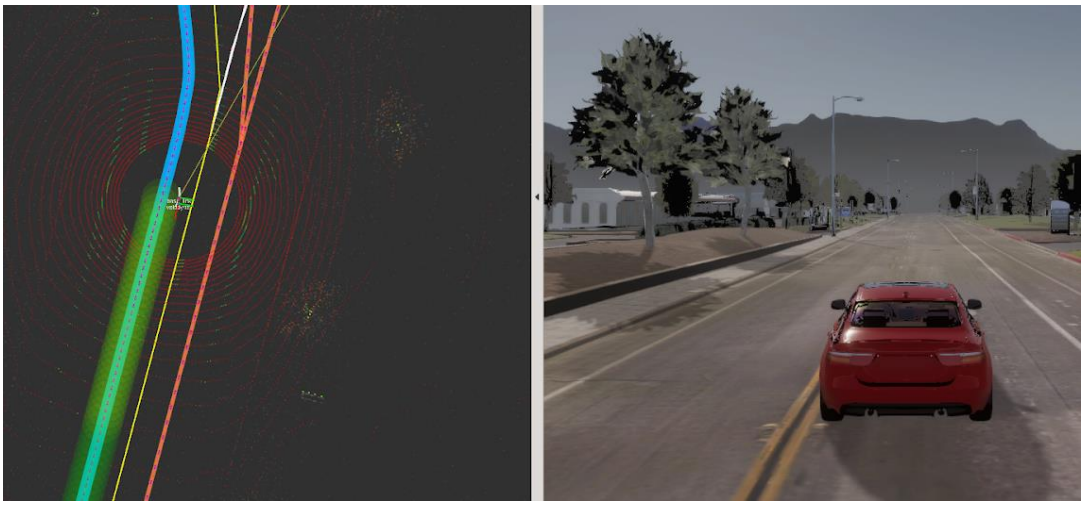

Figure 13: First Oscillation with Tuned Follower

More testing was done with varying the parameters of the pure pursuit follower to increase velocity and lookahead distance, but the computation for the LiDAR localization was not fast enough to maintain localization at moderate speeds. An attempt to improve localization accuracy was made by fusing GNSS data with the LiDAR localization, but being a relatively new feature in Autoware, these tests were not able to be completed because of software issues.

Even with the improvement, the default algorithm used for the demo proved to be insufficient for realistic path following. However, more precise controls and algorithms can be developed into Autoware for path following, so the lacking performance with the default modules did not discredit Autoware's functionality. The purpose of this demonstration was to uncover the functionality and potential of the software, and with the ease of implementing the Autoware platform, this demonstration showed that, with development, Autoware can be an effective solution for prototyping automated vehicles.

3.3 Unity Development

The first of three scenarios created in Unity was the straight crossing paths at a non-signalized junction scenario. An overhead of the scenario can be seen in Figure 14.

Figure 14: Straight Crossing Paths at Non-Signalized Junction

The red vehicle at the bottom of the figure is the subject vehicle (SV), and the white vehicle at the left side of the intersection is the primary other vehicle (POV). In this scenario, the POV is programmed to drive into the intersection for a collision with the SV impacting the right side of the POV. The collision was timed based on the velocity of the SV so that the POV can account for variability in the speed profile of the SV to always be timed correctly for collision. A tuner was also created for the user to adjust how far into the intersection the POV should be relative to the position of the SV allowing for testing SV response to a POV partly across the intersection or just

initiating a cross when the SV is in the intersection. Figure 15 below shows the collision point without tuning.

Figure 15: Intersection Scenario Collision without Tuning

The second scenario created was the following scenario with the SV located behind the POV with a set speed profile. This scenario can be found in Figure 16.

Figure 16: Vehicle Following Scenario

The functionality created in this scenario includes a variable endpoint, variable starting distance between the SV and the POV, and variable speed profiles. The primary profile used was a sudden lead vehicle stop with a deceleration variable that can be tuned to test functionality with gradual or sudden stops, but many other profiles can be readily integrated.

Next, a roundabout scenario was added. Multiple vehicles can be added to the roundabout with speed, distance from the center, and phase in the roundabout being the adjustable variables. With these adjustable parameters of the scenario, a large variety of complex test cases can be created. Two different roundabout configurations with are displayed below.

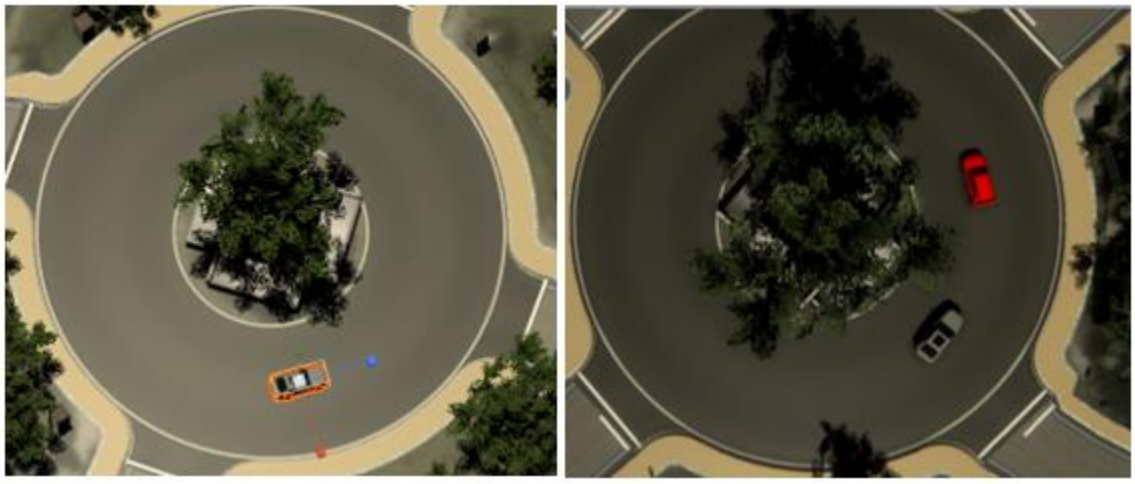

Figure 17: Roundabout Scenarios

Additionally, with more than one roundabout being present in the Linden route, a roundabout selector was created which allows the user to specify which roundabout the vehicle should be circling.

Finally, a pedestrian crossing scenario was created because of the high risk of the scenario and likelihood of occurrence because the Linden route is located in a residential area. Figure 18

below shows the general scenario with a pedestrian located on the sidewalk by the intersection highlighted and the SV approaching at the bottom.

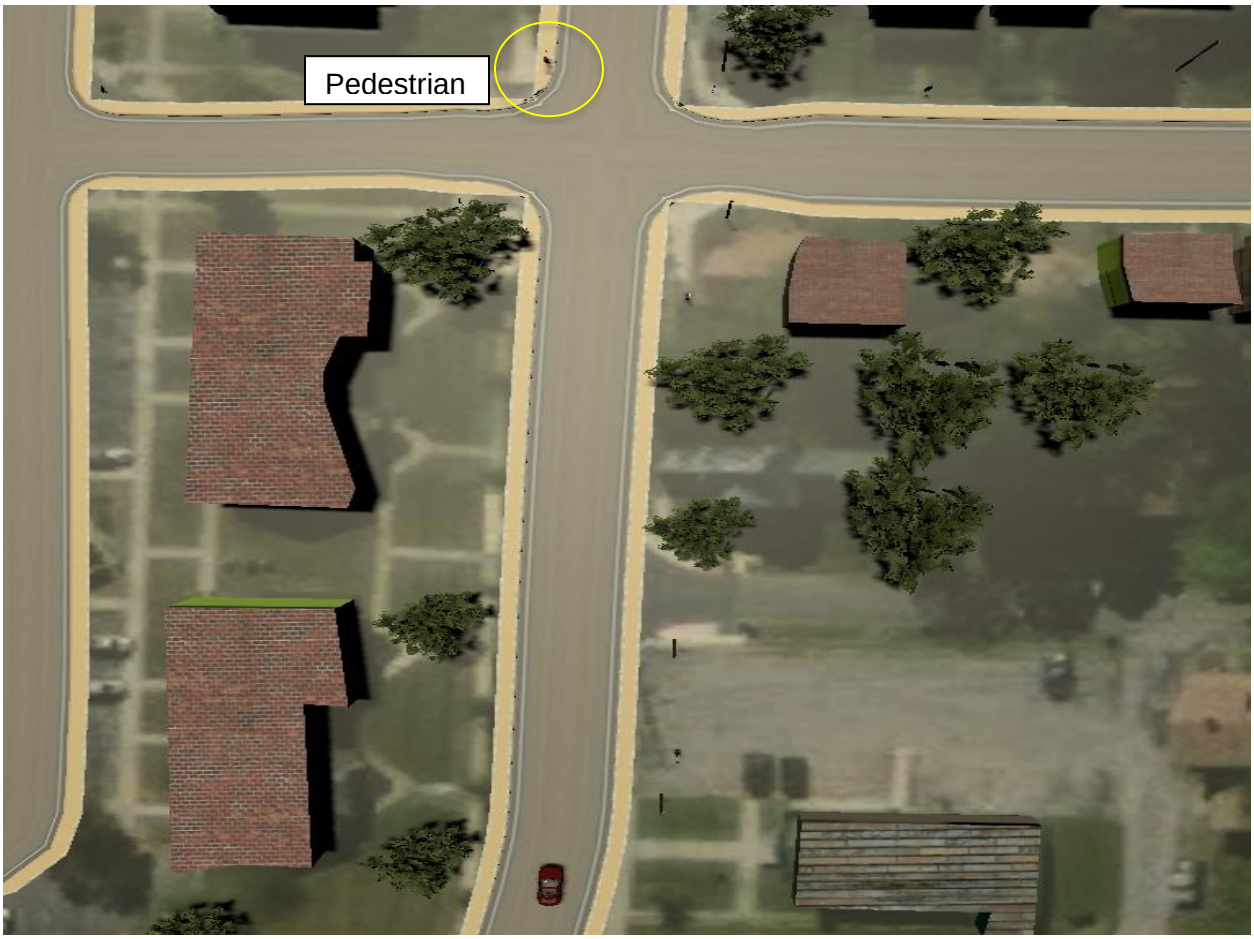

Figure 18: Pedestrian Crossing Scenario

Functionality created in this script was tuning for the rate that the pedestrian crossed the intersection for edge cases where the pedestrian may start to cross or where the pedestrian may be ending the cross when the SV is imminent. Animation was also added to the pedestrian for realistic simulation of the sensing. The collision point with the pedestrian in this scenario can be seen below.

Figure 19: Pedestrian Collision

More generally, a parameter for adjustable friction, both in the longitudinal and lateral directions independently, was created for the SV. This flexible parameter allows for scenario testing in extreme scenarios and adds to the accuracy of the simulator.

Overall, each of the scripts that was created was made to be as dynamic as possible and easily configurable to a variety of scenes and scenarios. Documentation for each individual script was also created for enhanced clarity and ease of use for future development or adaptations.

Chapter 4: Conclusion and Future Work

4.1 Conclusions from Study

From the work performed in this project, Autoware's decision maker was analyzed with respect to the information it processes and how it coordinates with the state machine. The primary demonstration of the decision maker was conducted through testing of readily available data from Autoware's GitHub repository. Additional testing with Autoware running with LGSVL Simulator demonstrated the potential of Autoware in general as well as the potential of simulations with Unity to develop customized maps and scenarios to test with. Simulation environments in Unity were created with specific test cases of statistically probable scenarios for an autonomous shuttle to encounter on the Linden route. The base scenarios were created to be expanded and customized to many more cases.

4.2 Future Work

Work remains to be done with integrating the developed Unity scenes with Autoware. The main difficulty in this lies in creating point cloud and vector maps for the scenes. A point cloud map of the portion of the Linden route used for the scenario testing was created with Autoware's NDT mapping tool, but the point cloud and vector maps are not aligned which requires further development before the full potential of the Unity scenes can be accessed. This problem that was not resolved is displayed in Figure 20 which is an RViz visualization of the point cloud and vector map data.

Figure 20: Point Cloud Map and Vector Map Misalignment

The white shape on the left of the figure is the point cloud map while the orange lines in the top right are the vector map created from the Unity scene. Initial diagnostics including relocating the Unity map origin and relocating the position of the vector map tool were attempted with no improvement, so this remains a problem with the ongoing testing. Once the maps are aligned, the scenarios can be integrated effectively with Autoware and adjusted to a large variety of different cases.

References

[1] Edelstein, Stephen. “2019 Audi A8 Won't Get Traffic Jam Pilot in the United States.” Digital Trends, Digital Trends, 18 May 2018, www.digitaltrends.com/cars/2019-audi-a8-traffic-jam-pilot-not-coming-to-us/.

[2] Walker, Jon. “The Self-Driving Car Timeline – Predictions from the Top 11 Global Automakers.” Emerj, Emerj, 19 Feb. 2019, emerj.com/ai-adoption-timelines/self-driving-car-timeline-themselves-top-11-automakers/.

[3] “CARMA Platform.” CARMA Platform | FHWA, Federal Highway Administration, highways.dot.gov/research/operations/CARMA-Platform.

[4] S. Kato et al., "Autoware on Board: Enabling Autonomous Vehicles with Embedded Systems," 2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS), Porto, 2018, pp. 287-296.

[5] Becker, Jan. “Announcing the Autoware Foundation - Open Source for Autonomous Driving.” Medium, Medium, 9 Dec. 2018, medium.com/@jan_26255/announcing-the-autoware-foundation-open-source-for-autonomous-driving-f340e9960eda.

[6] “CPFL/Autoware-Manuals.” GitHub, Nagoya University, github.com/CPFL/Autoware-Manuals/blob/master/en/Autoware_UsersManual_v1.1.md.

[7] “AbangLZU/Autoware.” GitHub, github.com/AbangLZU/Autoware/tree/master/ros/src/computing/planning/decision/packages/decision_maker.

[8] Guvenc, L., Aksun-Guvenc, B., Li, X., Arul Doss, A.C., Meneses-Cime, K.M., Gelbal, S.Y., 2019, Simulation Environment for Safety Assessment of CEAV Deployment in Linden, Final Research Report, Smart Columbus Demonstration Program – Smart City Challenge Project (to support Contract No. DTFH6116H00013).

[9] Li, X., Arul Doss, A.C., Aksun-Guvenc, B., Guvenc, L., 2020, “Pre-Deployment Testing of Low Speed, Urban Road Autonomous Driving in a Simulated Environment,” WCX20: SAE World Congress Experience, April 21-23, Detroit, Michigan, Session AE100 ADAS and Autonomous Vehicle Systems, SAE Paper Number: 2020-01-0706.

[10] W. Najm, J. Smith, M. Yanagisawa. (2007). *Pre-Crash Scenario Typology for Crash Avoidance Research.* U.S. Department of Transportation, NHTSA.

[11] Ami&, 0. “Integrated Mobile Robot Control.”, Masters -is, Dept Of EleCnical ad Computer Engineering, CMU, May, 1990.

Appendix

Unity Scene Scripts

DPosReset.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

//***Resetting POV Position***

//Variables to adjust are Tuner and which key to press for reset
//Set to LEFT ALT currently

public class DPosReset : MonoBehaviour {

  public string key = "left alt";
  public float disTuner = 0f;
  Vector3 originalPos;
  Vector3 TunePos;

  void Start () {
    originalPos = transform.position;
    TunePos = originalPos;
  }

  void FixedUpdate(){

    if(Input.GetKeyDown(key)){
      TunePos.z = originalPos.z+disTuner;
      transform.position = TunePos;
    }
  }
}
```

PosMatch.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

//***Intersection Scenario Timing***//

//Variable to adjust is tuner for collision timing

public class PosMatch : MonoBehaviour {
  Vector3 originalPos;
  Vector3 newPosition;
  Vector3 EgoPos;
  public Rigidbody rb;
  float dz;   //relative z position of ego
  float POVDis;   //Distance of POV to intersection
  float EgoDis;   //Distance of ego to intersection
  public float tuner = 1f;    //tune timing of collision, (set as 2.5 for pedstrian)

  void Start () {
    EgoPos = GameObject.Find("ADL_vehicle_test").transform.position;

    originalPos = transform.position;
    newPosition = originalPos;

    //Setting relative distances
    POVDis = Mathf.Abs(originalPos.x - EgoPos.x);
    EgoDis = Mathf.Abs(originalPos.z-EgoPos.z);

  }

  // Update is called once per frame
  void FixedUpdate () {
    //Position of Ego in lane
    dz = EgoPos.z - GameObject.Find("ADL_vehicle_test").transform.position.z;

    //New Position with relative distance
    newPosition.x = transform.position.x - POVDis*dz/EgoDis*tuner;
    rb.velocity = (newPosition-transform.position)/Time.deltaTime;
```

```
      //Resetting Ego pos for next calc
      EgoPos = GameObject.Find("ADL_vehicle_test").transform.position;
   }
}
```

VelocitySet.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

//***Velocity Set Script***//

//Variables to adjust are Decel
//Variables to set are rb
//Velocity Profile can be changed in FixedUpdate

public class VelocitySet : MonoBehaviour {

   public Rigidbody rb;
   public float speed = 0.1f;
   public float xadj = 0.07f; //So car stays in center of lane
   public float accel = 2f;
   public float stopTol = 10f; //velocity tolerance for stopping so when it gets small enough just
set to zero
   public string Start_Key = "b";
   int state = 0;
   float End = 90;     //Ending z value (specific to location on map so needs to change if location
changes)
   public float decel = 0.015f;    //deceleration rate

   void FixedUpdate () {

      //SETTING VELOCITY PROFILE

      //Initialization
      if(Input.GetKeyDown(Start_Key)){
```

```
            state =1;
        }

        if(state ==0){
            rb.velocity = new Vector3(0,0,0);
        }

        //Constant starting acceleration
        if(state ==1){
            rb.velocity = rb.velocity + new Vector3(-accel*xadj,0,-accel)*Time.deltaTime;

            if(Mathf.Abs(rb.velocity.z)>speed/Time.deltaTime){
                state =2;
            }
        }

        //Constant velocity
        if(state==2){
            rb.velocity = new Vector3(-xadj*speed,0,-speed)/Time.deltaTime;

            if(transform.position.z < End){
                state =3;
            }
        }

        //Constant Deceleration
        if(state==3){

            if(rb.velocity.z<=-stopTol){
                rb.velocity = rb.velocity - rb.velocity*decel;
            }
            else{
                rb.velocity = new Vector3(0,0,0);
                state = 0;
            }
        }
        Debug.Log(state);
    }
}
```

RoundaboutDriving.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;


//***NPC Driving around a roundabout***

//Adjustable variables: speed, distance from center, number of vehicles, vehicle index and phase
//Currently max vehicles supported is 2 and must change vehicle index between vehicles

//Only parameters to change are Roundname and rb
//Parked car prefab was used with added Rigidbody and Box collider
//Zero Friction was set for material on Box Collider
//Y position and X,Z Rotation were frozen in Rigidbody
public class RoundaboutDriving : MonoBehaviour {

  public string Roundname = "Brook_roundabout";
  public int numVeh = 1;  //specify how many vehicles in roundabout
  public int vehidx = 1;
  public Rigidbody rb;    //Rigidbody of POV
  public float Dist = 13f;    //Gain for setting distance
  public float speedset = 50f;    //Gain for setting speed
  float timeCounter = 0;
  Vector3 newPos;
  Transform center;
  public float phase;
  float phased;

  void Start () {
     center = GameObject.Find(Roundname).transform;          //Initializes center of roundabout

     //Sets phase of vehicles
     if(numVeh ==2 && vehidx ==2){
        phase = Mathf.PI;
        transform.rotation = Quaternion.AngleAxis(180,Vector3.up);
     }
     else{
        phase = 0;
```

```
    }
  }

  void FixedUpdate () {

    timeCounter +=Time.deltaTime;   //Used for oscillations
    phased = phase*180/Mathf.PI;    //used for rotation

    float speed = speedset*Time.deltaTime;

    //creates circular motion
    float x = Mathf.Cos(timeCounter*speed + phase)*Dist;
    float y = 0;
    float z = Mathf.Sin(timeCounter*speed + phase)*Dist;

    newPos = new Vector3 (x,y,z)+center.position;
    rb.velocity = (newPos-transform.position)/Time.deltaTime;

    //rotates orientation of vehicle
    transform.rotation = Quaternion.AngleAxis(-timeCounter*speed*180/Mathf.PI-
phased,Vector3.up);
  }
}
```

PedCrossing.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

//***Used to animate Pedestrian Crossing into Road ***

//Used in conjunction with PosMatch.cs to time pedestrian crossing into road
public class PedCrossing : MonoBehaviour {

  public Animator anim;
  // Use this for initialization
  void Start () {
    anim = gameObject.GetComponent<Animator>();
```

```
  }

  // Update is called once per frame
  void Update () {
     anim.Play("Walk");
  }
}
```

Friction.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

//***Add this component as adjustable parameter to parent of the wheels***

public class Friction : MonoBehaviour {

  public float forward = 0.4f;
  public float sideways = 0.4f;
}
```

GetFric.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

//***Getting Friction***//

//Used by individual wheels to get friction set in parent

public class GetFric : MonoBehaviour {

  public float forwardc;
  public float sidewaysc;
  public WheelCollider Wc;       //Wheel collider of wheel to set friction
```

```
    void Update () {

        //Getting Friction from parent
        forwardc = GetComponentInParent<Friction>().forward;
        sidewaysc = GetComponentInParent<Friction>().sideways;

        //Setting Friction
        WheelFrictionCurve wf;
        wf = Wc.forwardFriction;
        wf.extremumSlip = forwardc;
        Wc.forwardFriction = wf;

        wf = Wc.sidewaysFriction;
        wf.extremumSlip = sidewaysc;
        Wc.sidewaysFriction = wf;

    }
}
```